

4 de julio de 2017

# Ingeniería del Software I

Samuel García  
Ignacio Ballesteros  
Pablo Conde de la Mata  
Borja Lozano  
Víctor Villar

# Índice general

<b>1. Introducción a la Ingeniería del Software</b>	<b>3</b>
1.1. La crisis del Software . . . . .	3
1.2. Costes en la Ingeniería del Software . . . . .	4
1.3. El Software . . . . .	5
1.4. Procesos en el Software . . . . .	5
1.5. Ciclo de Vida . . . . .	6
<b>2. Ingeniería de Requisitos</b>	<b>8</b>
2.1. Requisitos . . . . .	8
2.2. Análisis de Requisitos . . . . .	10
2.3. Especificación de Requisitos . . . . .	11
<b>3. Diseño Estructurado de Alto Nivel</b>	<b>14</b>
3.1. Introducción . . . . .	14
3.2. Análisis vs Diseño . . . . .	14
3.2.1. Enfoque de Yourdon . . . . .	15
3.3. Notaciones . . . . .	15
3.3.1. Diagramas de Flujo de Datos . . . . .	16
3.3.2. Diccionario de Datos . . . . .	16
3.3.3. Especificaciones de proceso . . . . .	17
3.3.4. Entidad/Relación . . . . .	17
3.4. Relación entre notaciones . . . . .	18
3.5. Proceso de Construcción . . . . .	19
3.5.1. Construcción del Modelo Ambiental . . . . .	20
3.5.2. Construcción de un Modelo de Comportamiento preliminar .	20
3.5.3. Acabado del Modelo . . . . .	21
<b>4. Objetos</b>	<b>22</b>
4.1. Orientación a objetos . . . . .	22
4.1.1. Enfoque estructurado . . . . .	22
4.1.2. Enfoque orientado a objetos . . . . .	23

4.1.3.	Objetos . . . . .	24
4.1.4.	Representación gráfica de los objetos . . . . .	28
4.1.5.	Clases . . . . .	30
<b>5.</b>	<b>Arquitectura</b>	<b>36</b>
5.1.	Introducción a la arquitectura . . . . .	36
5.2.	Requerimientos . . . . .	36
5.3.	Diseño de estructuras . . . . .	37
5.4.	Diseño de Arquitecturas . . . . .	38
5.5.	Documentación . . . . .	38
5.6.	Evaluación . . . . .	38
5.7.	Implementación . . . . .	39
5.8.	Arquitectura Software . . . . .	40
5.8.1.	Estilos . . . . .	40
5.8.2.	Índice de un documento de arquitectura . . . . .	40
5.8.3.	Patrones . . . . .	41
<b>6.</b>	<b>Apéndices</b>	<b>42</b>
6.1.	Examen Junio 2009 . . . . .	42

# Bloque 1

## Introducción a la Ingeniería del Software

**Ingeniería del Software** (*IEEE*) La aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento de *software*.

Las **Ciencias de la Computación** se preocupa de los fundamentos de la teoría; mientras que la Ingeniería del Software de los aspectos prácticos.

La **Ingeniería del Software** estudia los **productos** producidos (ejecutables, módulos, sistemas, librerías...) y los **procesos** usados para producir esos productos.

### 1.1. La crisis del Software

Los principales problemas que hay detrás de la crisis del Software son:

- El incremento en el tamaño y la complejidad.
- Los sobrecostos.
- Fallos en el diseño.
- Mal mantenimiento.
- Herramientas no solo de programación.

Pero desde un enfoque más moderno, también se aprecian otros tipos de problemas:

- Falta de robustez en el Software para componentes críticos o de los que somos dependientes.
- Excesiva complejidad en el Software como para entenderlo y comprenderlo.
- Exigencia de cambiar rápidamente.

Frente a esto, desde la Ingeniería del Software se plantea la mejora en las metodologías y el uso de lenguajes de alto nivel (mayor abstracción).

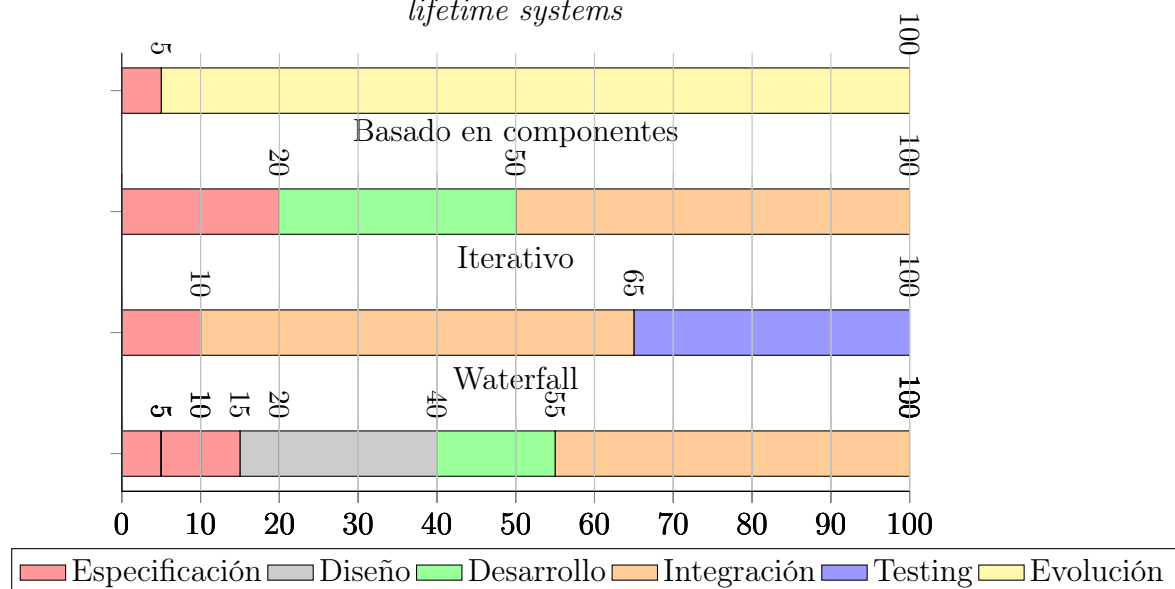
Tras el intento de solucionar estos problemas, el concepto de Ingeniería del Software se redefine en:

*Disciplina que tiene como objetivo la producción de Software libre de errores, sin retrasos y dentro del presupuesto; satisfaciendo las necesidades del cliente*

## 1.2. Costes en la Ingeniería del Software

En cuanto a la realización del producto, aproximadamente el 60 % del coste es en desarrollo, y el 50 % en pruebas (test). Sin embargo, con el uso del software, los gastos de mantenimiento superan a los del desarrollo.

Figura 1.1: Distribución del coste según metodología *lifetime systems*



Cuanto más tarde se encuentran errores en el Software, o más tarde se hace un cambio de requisito, los cambios son más costosos. Comparando el cambio en dos fases del ciclo de vida:

Temprano	Tardío
Cambio en la especificación.	Cambio en la especificación.
	Cambiar el código y la documentación.
	Probar el cambio.
	Testing.
	Instalación del producto en el cliente.

A medida que pasa el tiempo, la cantidad de fallos que aparecen aumentan debido al deterioro.

### 1.3. El Software

Entre las características que se le atribuyen a un *producto Software* encontramos:

- Múltiples **programas**.
- Archivos de **configuración**.
- La **documentación** del sistema.
- La documentación de **uso**.
- Los **datos** del sistema.
- **Actualización** de información.

El *Software* se realiza para **clientes particulares** o para el uso **general**. Esto está también relacionado con que los productos de software sean **genéricos** o **a medida**.

Dependiendo del producto desarrollado, el software puede entrar en categorías como: tiempo real, negocios, científico, emebido, PC, IA, Web...

Los atributos de un **buen Software** varían según las perspectivas:

Usuario	Desarrolador
Exactitud	Consistencia
Confiabilidad	Comprensibilidad
Eficiencia	Capacidad de ser probado
Mantenibilidad	Compacidad
Usabilidad	Compatibilidad
Robustez	

### 1.4. Procesos en el Software

Se entiende por *proceso Software* un conjunto de **actividades y resultados** asociados a la producción de Software.

Este proceso se puede analizar desde diferentes perspectivas:

- Flujo de **trabajo**.
- Flujo de **datos**.
- **Acción**.

Los modelos del *ciclo de vida* del Software especifican las fases del *proceso de Software*. Hemos mencionado ya ejemplos en la sección 1.2. Un modelo está compuesto de:

- Descripción propia.
- Reglas.
- Recomendaciones (*guías de estilo*).
- Procesos (*actividades a seguir*).

Estos **modelos** están orientados a resolver los retos de la ingeniería del Software, muy relacionados con los atributos del buen Software indicados en la sección 1.3.

- Heterogeneidad de plataformas.
- Entrega más rápida.
- Confianza.
- Gastos en el Hardware/Software.
- Adaptabilidad a nuevas tecnologías.
- Usabilidad.
- Mantenimiento.

## 1.5. Ciclo de Vida

Qué hacer →Cómo hacerlo →Hacerlo →Probarlo →Usarlo →Mantenerlo

Para la realización del Software tendremos que tener en cuenta:

- Escala.
- Productividad.
- Calidad (ISO).
  - Funcionalidad, Fiabilidad, Usabilidad, Eficiencia, Mantenibilidad, Portabilidad.
- Consistencia.
- Tasa de cambio.

Este ciclo de vida se organiza en fases. De cada fase se obtendrá un resultado que se utilizará en las siguientes fases. El *ciclo de vida* del Software se enfoca en manejar la complejidad y el cambio a lo largo de un proceso.

Existen distintos *ciclos de vida*, nombrados como modelos (sección 1.3):

- Informal.
- Convencional.
- Incremental.
- Evolutivo.
- Prototipado (puede ser incluido en los anteriores modelos).

Estos modelos tienen un equilibrio entre los siguientes factores:

- Velocidad de desarrollo.
- Calidad.
- Visibilidad.
- Sobrecarga de gestión.
- Exposición al riesgo.
- Relaciones públicas.



# Bloque 2

## Ingeniería de Requisitos

La IR trata de los principios, métodos, técnicas y herramientas que permiten descubrir, documentar y mantener los requisitos para sistemas basados en computadora, de forma sistemática y repetible.

### 2.1. Requisitos

Definición formal de Jackson y Zave:

Todo problema software consiste en configurar una máquina para que ejerza unos efectos  $R$  en un dominio  $K$ . La conexión de la máquina con el dominio se realiza a través de una interfaz  $S$ .

- Los efectos  $R$  son los requisitos: Necesidades, metas, objetivos. Expresan ideas que no son una realidad ahora mismo, pero lo serán en un futuro, una vez que el sistema esté en funcionamiento.
- El conocimiento del dominio  $K$  describe el contexto. Expresa realidades actuales, cosas que son verdad independientemente de la existencia o inexistencia del sistema.
- La máquina ( $hw+sw$ ) es la que realizará los requisitos  $R$ , gracias a  $S$ , que describe la conexión con el dominio  $K$ . Por tanto,  $S$  describe el comportamiento externo (observable) del sistema.
- Idealmente:  $K$  y  $S \implies R$

Pirámide de **Leffingwell-Widrig**:

- En primer lugar, las necesidades (needs), que son objetivos a muy alto nivel del producto, y no debería haber más de 3 ó 4.
- Las features son una descripción breve, en lenguaje del usuario, de las capacidades de alto nivel de un producto.

- Finalmente, los requisitos software, en la base de la pirámide, son descripciones más técnicas acerca de cómo el producto realiza las features.

## Requisitos funcionales y no funcionales

Los requisitos **funcionales** describen los servicios mientras que los requisitos no funcionales son restricciones sobre los requisitos funcionales.

Tipos de requisitos **no funcionales** (definiciones informales):

**Fiabilidad (reliability)** Es el grado con que un sistema cumple, o no, sus requisitos.

**Seguridad (security)** Hace referencia a la seguridad frente a ataques por parte de humanos o software malicioso.

**Seguridad (safety)** Hace referencia a la incapacidad del sistema de provocar accidentes.

**Usabilidad** Grado de integración del sistema en las tareas que realiza el usuario.

**Robustez** Capacidad del sistema de resistir situaciones extremas de todo tipo.

**Disponibilidad (availability)** Hace referencia al tiempo durante el cual el sistema no es utilizable.

**Rendimiento (performance)** Capacidad de hacer las tareas con el menor consumo de recursos posible.

**Tiempo de respuesta** Latencia entre una solicitud y la respuesta dada por el sistema.

**Capacidad** Cantidad de usuarios que pueden utilizar el sistema al mismo tiempo.

**Throughput** Cantidad de datos o transacciones procesadas por unidad de tiempo.

**Testabilidad** Capacidad de ser probado con facilidad.

**Portabilidad** Capacidad de ser transformado con facilidad a otra plataforma hw/sw.

## Proceso de Requisitos

### Edución de Requisitos

Se orienta a la captura y descubrimiento de los requisitos. Para ello se identifica a los interesados (stakeholders<sup>1</sup>) y se establecen las primeras relaciones entre ellos y el equipo de desarrollo.

Los requisitos pueden proceder de:

- Metas.
- Conocimiento del dominio de la aplicación. Interesados: Afectados por el cambio.
- El entorno físico que rodea al sistema.

---

<sup>1</sup>Un stakeholders es toda aquella persona que se ve afectada por la existencia del futuro sistema

- La organización.

Técnicas de educación:

- Preguntas Libres de Contexto.
- Brainstorming.
- Creatividad.
- Entrevistas.
- Observación y análisis de tareas.
- Casos de uso / Escenarios.
- Prototipado.

## 2.2. Análisis de Requisitos

El Análisis de Requisitos consiste en detectar y resolver conflictos entre los requisitos identificados, en precisar los límites del sistema, precisar la interacción con su entorno, trasladar los requisitos de usuario a requisitos implementables en software, etc. Se realizan tres (sub)tareas fundamentales: **Clasificación, Modelización y Negociación**.

### Clasificación

Se clasifican en funcionales / no funcionales:

- Por prioridades.
- Por coste de implementación.
- Por niveles (alto nivel, bajo nivel).
- Según su volatilidad/estabilidad. Los volátiles son aquellos que posiblemente pierdan su razón de ser en el futuro, normalmente debido a cambios en políticas o en los procesos de gestión.
- En requisitos sobre el proceso o sobre el producto.
- Además, en Ingeniería de Sistemas, se realiza la ubicación de requisitos (Requirements Allocation), dónde se decide qué va a realizar el software y qué va a realizar el hardware.

### Modelización

Ciertos aspectos de los requisitos se expresan mejor mediante modelos de datos, de control, de estados, de interacción, de objetos, etc.

## Negociación

Es durante el análisis cuando se descubren muchos de esos conflictos. EL CONFLICTO NO ES RECHAZABLE y no debe resolverse por decreto, sino mediante un proceso de negociación. Desde este punto de vista, los conflictos son positivos, pues SON FUENTE DE NUEVOS REQUISITOS. Los acuerdos alcanzados deben ser convenientemente anotados, favoreciéndose así la trazabilidad de los requisitos a sus orígenes.

## 2.3. Especificación de Requisitos

La especificación establece el almacenamiento de requisitos en algún medio (electrónico o no) que permita su distribución, gestión, impresión, búsquedas, etc.

El documento es el modo habitual de guardar y comunicar requisitos. Es buena práctica utilizar, al menos, dos documentos, a distinto nivel de detalle.

**DRU, Documento de Requisitos de Usuario (URD)** El DRU se escribe desde el punto de vista del usuario/cliente/interesado. Normalmente los requisitos de usuario, contenidos en la DRU, no poseen demasiado nivel de detalle (serían features, en su mayor parte). Se incluye la descripción del problema actual (razones por las que el sistema de trabajo actual es insatisfactorio) y las metas que se espera lograr con la construcción del nuevo sistema.

**ERS, Especificación de Requisitos de Software (SRS)** La ERS desarrolla mucho más los contenidos de la DRU. Los requisitos del software contenidos en la ERS son, por tanto, más detallados.

Llegado el momento de elaborar un documento en papel o, dicho de otra forma, llegado el momento de volcar en papel un subconjunto de los requisitos almacenados, de manera que las partes implicadas puedan firmarlo a la manera de un contrato, hay varias formas de organizar tal documento. Lo normal es seguir los estándares del IEEE.

Una ERS de calidad debería poseer, entre otros, los siguientes atributos:

- No ambigua.
- Completa.
- Correcta.
- Comprensible.
- Verificable.
- Consistente.
- Independiente del diseño.
- Anotada por importancia relativa.

- Anotada por estabilidad relativa.
- Anotada por versión.
- Expresada a distintos niveles de abstracción.
- Precisa.
- Trazable.
- Trazada.
- Con referencias cruzadas.

## Validación de Requisitos

El objetivo de la validación es descubrir problemas en el subconjunto de requisitos que han sido seleccionados para su inclusión en la ERS, antes de comprometer recursos a su implementación.

Durante la validación, el documento debe revisarse para descubrir omisiones, conflictos, ambigüedades, y para comprobar su calidad y su grado de adhesión a estándares. La fórmula más empleada para validación son las llamadas Revisiones de Validación (Reviews). En ellas, un grupo de personas (incluyendo usuarios) se ocupan de revisar el documento de requisitos para encontrar problemas. Estas reviews no se hacen a ciegas, pues se suelen utilizar listas de comprobación (checklists) que nos dicen qué es lo que hay que buscar.

## Gestión de Requisitos

La Gestión de Requisitos consiste en gestionar los cambios a los requisitos. Esto implica:

- Definir procedimientos de cambios.
- Cambiar los atributos de los requisitos afectados (prioridad, versión, etc.).
- Mantener la trazabilidad.
- Control de versiones del documento de requisitos.

Hay dos conceptos importantísimos en Gestión de Requisitos: la **trazabilidad** y las **líneas base**.

La trazabilidad hace referencia al establecimiento de enlaces entre unas cosas y otras. En la Gestión de Requisitos se suelen manejar tres tipos de trazabilidad<sup>2</sup>:

**Trazabilidad hacia atrás** Dado un requisito, nos permite conocer su origen.

**Trazabilidad hacia delante** Dado un requisito, podemos conocer qué artefactos generados durante el proceso software se encuentran relacionados con él.

**Trazabilidad interna** Dado un requisito, la trazabilidad interna indica qué otros requisitos se encuentran relacionados con él.

---

<sup>2</sup>Preguntado en el examen de 2017

Una línea base (**baseline**) es un conjunto de requisitos que, mediante acuerdo entre las partes implicadas, se ha decidido no modificar.

## Bloque 3

# Diseño Estructurado de Alto Nivel

### 3.1. Introducción

Podemos encontrar dos paradigmas principales para la construcción de sistemas software. Por un lado, tenemos el **Estructurado** y por otro el **Orientado a Objetos**. Asimismo, el Diseño Estructurado se basa en tres fundamentos:

- *Top-Down*
- Sistemas como funciones.
- Datos y funciones separados.

### 3.2. Análisis vs Diseño

Los lenguajes de alto nivel en el pasado son los lenguajes de bajo nivel del futuro<sup>1</sup>, así mismo las técnicas de **análisis** de ayer son técnicas de **diseño** de hoy.

Los problemas reales de los usuarios son problemas de organización, monitorización, control, etc; Por tanto los problemas no son ni *estructurados* ni *orientados a objetos*, pero las **soluciones** sí.

El análisis pretende comprender algo que **ya existe**: estudiarlo, delimitarlo, clasificarlo...

Son tareas del análisis, por ejemplo:

- Realizar un estudio del mercado antes del lanzamiento de un producto.
- Encontrar las causas del fracaso de un proyecto software.
- Estudiar las características del terreno donde se construirá un edificio.

---

<sup>1</sup>Pues eso, funcional.

En general se encarga de comprender un problema antes de resolverlo (**requisitos**).

El diseño busca crear **algo que no existe**: elaborar un plan antes de comenzar a construir.

Ejemplos de tareas de diseño son:

- **Crear** un producto para lanzarlo al mercado.
- **Elaborar** los planos de un edificio.

En general el diseño se encarga de utilizar cierta tecnología para resolver un problema (**DFDs, OO**).

En Ingeniería de Software se denominaba “*Análisis*” de problemas a lo que, en realidad, era un “*Diseño*” de soluciones.

### 3.2.1. Enfoque de Yourdon

*Nota: A pesar de que el libro de Yourdon se denomina “Análisis Estructurado Moderno” lo consideraremos como si fuera diseño. Su técnica no sirve para describir problemas, sino soluciones (tecnologías).*

- Se basa en una serie de notaciones bastante conocidas como **DFDs, E/R**, etc.
- Se mantiene una relación entre las notaciones.
- Su objetivo es elaborar un “**Modelo esencial**” del sistema.
- No es top-down, sino **middle-up, middle-down**.
- La construcción del Diagrama de Flujo de Datos está **dirigida por acontecimientos**.
- Pese a ser moderno, es **antiguo**, pero no tanto como el enfoque puramente *top-down*.

## 3.3. Notaciones

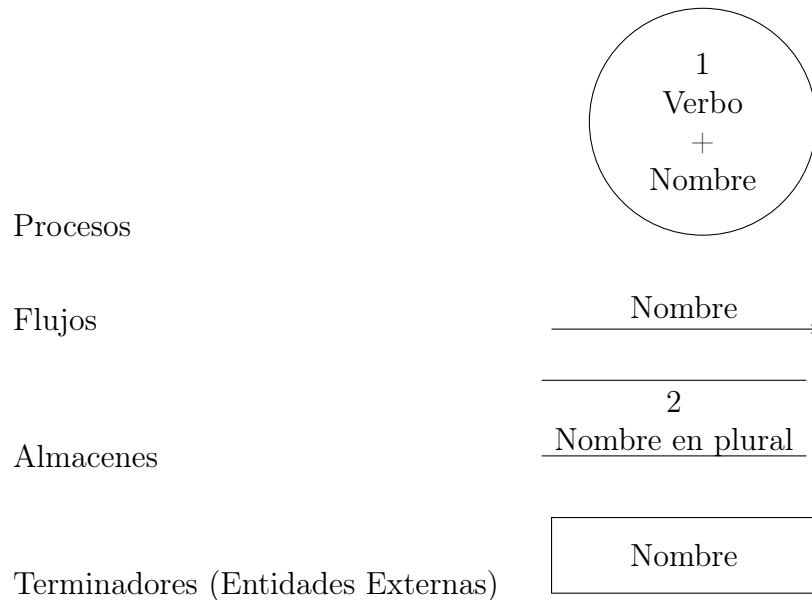
Las notaciones que describiremos son:

- Diagrama de Flujo de Datos (DFDs).
- Diccionario de Datos (DDs).
- Especificación de procesos.
- Entidad/Relación (E/R).



### 3.3.1. Diagramas de Flujo de Datos

La **notación** se escribe a continuación:



Las **reglas** son:

- Nombres con significado.
- Numerar los procesos NO implica secuencia.
- Evitar DFDs complejos.
- Todo DFD se redibuja varias veces.

La **regla del balanceo** mantiene la consistencia entre las entradas y salidas de una burbuja padre y las entradas y salidas de todas sus burbujas hijas.

Para preservar la **consistencia lógica** de un Diagrama de Flujo de Datos hay que **evitar**:

- Sumideros infinitos.
- Generaciones espontáneas.
- Flujos y procesos no etiquetados.
- Cuidado con los almacenes de sólo lectura o sólo escritura.

### 3.3.2. Diccionario de Datos

Un Diccionario de Datos es una **lista organizada** de todos los datos, que describe el **significado** y la **composición** de los flujos y almacenes, así como también las **relaciones** entre almacenes mediante un E/R.

Su **notación** es:

Símbolo	Significado
=	<i>se define como</i>
+	<i>y</i>
()	opcional
{ }	iteración
□	separa alternativas
*...*	define datos elementales
**	comentario
@	identificador (clave)
	alternativa

### 3.3.3. Especificaciones de proceso

Son sólo para los procesos (burbujas) de más bajo nivel. Se puede utilizar:

- Lenguaje estructurado.
- Pre/post.
- Tablas de decisión.
- Gráficas.
- Lenguaje natural.
- Flow-charts.

**Presentación por niveles:**

- Los DFDs se presentan TOP-DOWN. Esto NO quiere decir que se construyan TOP-DOWN.
- El orden de presentación es: Diagrama de Contexto, Diagrama 0, Diagrama 1, etc.
- ¿Hasta donde?: Se detiene la descomposición cuando las burbujas pueden especificarse como procesos sencillos (en menos de una página).
- Sistemas sencillos: 2-3 niveles en total; medianos: 3-6; grandes: 5-8.

### 3.3.4. Entidad/Relación

Las **entidades**:

- Representan conjuntos de individuos.
- Cada individuo puede identificarse de manera única.
- Cada uno juega un papel necesario en el sistema.

Yourdon no utiliza atributos en el diagrama pero sí en el Diccionario de Datos. Las **relaciones** representan conjuntos de conexiones con sus cardinalidades.

Las **entidades asociativas** pueden verse como *entidades que contienen los atributos de una relación*. Se utilizan para representar relaciones acerca de las cuales se quiere guardar alguna información. Como por ejemplo:

*Un cliente compra artículos y queremos guardar información acerca del día y hora de la compra.*

En este caso “Día” y “Hora” serían **propiedades** de la relación, pero no de “Cliente” ni de “Artículo”.

En el Diccionario de Datos tendríamos:

- Cliente = @DNI + Nombre + Dirección + Telf.
- Artículo = @Código + Descripción + Precio.
- Compra = @DNI + @Código + Día + Hora.

A continuación se muestra un ejemplo de E/R con dos entidades (Persona y Herramienta) y una relación (Usa):



Las reglas para la construcción de un modelo E/R son las siguientes:

- Entrevistas, documentos: permiten la **identificación** de entidades.
- Se refinan y desarrollan en paralelo a los DFDs y DDs para mantener consistencia.
- Simplificar:
  - Eliminar objetos de instancia única.
  - Eliminar relaciones derivadas (calculables).
- Regla heurística: si no somos capaces de imaginar al menos 3 registros de una entidad, malo.

### 3.4. Relación entre notaciones

#### 1. Diagrama de Flujo de Datos y Diccionario de Datos

- Todo flujo y almacén del DFD deben definirse en el DD y viceversa.

#### 2. Diagrama de Flujo de Datos y Especificación de Proceso

- Toda burbuja del DFD debe asociarse con una burbuja del nivel inferior, o con una especificación de proceso pero **no ambos**.
- Toda especificación de proceso debe asociarse a una burbuja de bajo nivel.

- Las entradas y salidas deben coincidir.

### **3. Especificación de Proceso con Diagrama de Flujo de Datos y Diccionario de Datos**

- Los datos referenciados en la Especificación de Proceso deben coincidir con nombres de flujos o almacenes conectados a la burbuja, ser término local (variable local) o ser un componente del flujo o almacén.

### **4. Diccionario de Datos con Diagrama de Flujo de Datos y Especificación de Proceso.**

- Toda entrada del DD debe tener referencia en una Esp. Proc, un DFD o en el propio DD.

### **5. Entidad/Relación con Diagrama de Flujo de Datos y Especificación de Proceso.**

- Todo almacén del DFD debe corresponder a una entidad o entidad asociativa.
- Toda entidad, relación o entidad asociativa del E/R debe reflejarse en algún almacén del DFD.
- Los nombres deben coincidir: plural en DFD, singular en ER. Deben definirse ambas en el DD (ver a continuación).
- Las instancias de las entidades y las relaciones del ER deben ser creadas o eliminadas por algún proceso.
- Alguna burbuja del DFD define valores para cada componente de datos.
- Alguna burbuja del DFD usa los valores de cada componente de datos.

### **6. Entidad/Relación: extensiones al Diccionario de Datos.**

- En general, las entidades se nombran en singular y los almacenes en plural.
- Los atributos de las entidades se pueden especificar en el DD.

## **3.5. Proceso de Construcción**

Consiste en elaborar los dos componentes del llamado *Modelo Esencial*. Para lo que se construirá:

### 3.5.1. Construcción del Modelo Ambiental

- Escribir en un párrafo el **propósito del sistema**.
- Construir el **diagrama de contexto** (preliminar identificando las entidades externas).
- Construir una **lista de acontecimientos**.

Una *lista de acontecimientos* es una lista narrativa de los estímulos que ocurren en el mundo exterior a los cuales el sistema debe responder. Los hay de tres tipos:

- **Flujo**: se asocian con la llegada de algún flujo.
- **Temporal**: se disparan a una determinada hora.
- **Control**: (no los tratamos).

*Nota: los flujos y los acontecimientos son cosas distintas. No hay correspondencia uno-a-uno entre ambos.*

Los acontecimientos deben de contemplarse de fuera del sistema hacia dentro. Por ejemplo:

**Mal:** *El sistema recibe el pedido del cliente.*

**Bien:** *El cliente hace un pedido.*

La construcción de la *lista de acontecimientos* se realiza **a partir** de las *entidades externas*, pero el *Diagrama de Contexto* se puede hacer **a la vez** que la *Lista de acontecimientos*, se refinan mutuamente.

Los flujos del Diagrama de Contexto son el medio utilizado para la comunicación sistema-entorno. Los **flujos de entrada** son **utilizados** por los *acontecimientos* y los **flujos de salida** son **respuestas** a los *acontecimientos*.

- Todo **acontecimiento no temporal** debe tener **entradas** para que el sistema pueda detectarlo.
- Todo acontecimiento debe producir una **salida** inmediata y/o un **almacenamiento** de datos.

### 3.5.2. Construcción de un Modelo de Comportamiento preliminar

Consiste en desarrollar un *Diagrama de Flujo de Datos* y un diagrama *E/R preliminares*, además de las entradas iniciales del *Diccionario de Datos*. Se opone al enfoque clásico debido a que **no es** top-down.

- Se **dibuja** una burbuja o proceso por cada acontecimiento de la lista. El nombre de la burbuja describe la respuesta que se debe dar al acontecimiento.

- Se **dibujan** las entradas, salidas y almacenes apropiados (conectados a la burbuja).
- Se **conectan** las burbujas (de los acontecimientos) mediante almacenes.
- En **paralelo** se contruye el diagrama *E/R*.

### 3.5.3. Acabado del Modelo

- Para finalizar el modelo se debe realizar una **Nivelación ascendente** hasta llegar al Diagrama de Contexto.
  - Los procesos o burbujas se agrupan por respuestas relacionadas.
  - Según se asciende se ocultan los almacenes internos a una agrupación de procesos. *Sugerencia: hacer agregados de 7 +- 2 burbujas.*
- Puede requerirse nivelación ascendente.
- Completar el Diccionario de Datos.
- Completar la especificación del proceso (al final).
- En **paralelo** acabar el diagrama E/R.

# Bloque 4

## Objetos

### 4.1. Orientación a objetos

#### 4.1.1. Enfoque estructurado

Se denomina enfoque estructurado a la forma de pensar el software en términos de funciones de transformación de datos (se disocia entre funciones y datos, y las tareas se interpretan como una transformación de los últimos).

#### Ejemplo: Pintar un círculo

El enfoque estructurado resuelve el problema de pintar un círculo de la siguiente forma:

- Usa una definición de círculo que esté acorde con los recursos de software (en este caso la expresión algebraica).

$$R^2 \leq (x - x_0)^2 + (y - y_0)^2 \quad (4.1)$$

donde el radio  $R$  y las coordenadas del centro son las constantes que especifican un círculo concreto.

- Disocia la definición de círculo en dos partes y las reinterpreta:
  - Considera que  $R$  y el centro son datos para pintar el círculo y añade el color.
  - Convierte la expresión declarativa en una función operativa que transforma el conjunto de datos precedentes en  $(x, y, \text{color})$  de todos los píxeles para pintar el círculo en la pantalla.

- Como resultado final se obtiene un sistema capaz de pintar un círculo en términos de un proceso de transformación de datos.

El sistema software se expresa como una función  $F(x)$  que transforma el conjunto de datos  $(R, x_0, y_0)$  en otro conjunto de datos, en este caso de píxeles.

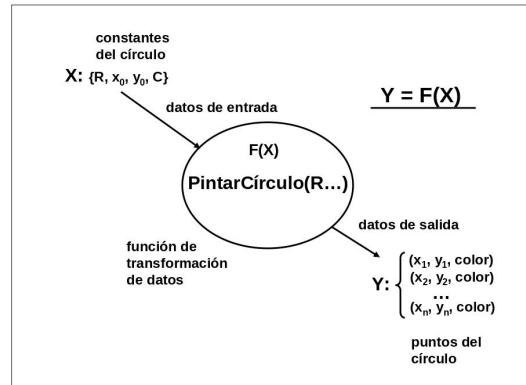


Figura 4.1: Sistema software como función de transformación de datos.

A este tipo de esquema se le denomina *diagrama de flujo de datos*. **El diagrama de flujo de datos es un esquema asíncrono** (no expresa secuencias); las flechas sólo indican los flujos de datos, no el orden de ejecución.

El principal problema del enfoque estructurado es latente en el momento en el que queremos añadir más elementos e interactuar con ellos, por ejemplo, pintar varios círculos y actuar sobre los mismos de forma selectiva, digamos borrar el segundo que se pintó. Podríamos hacer un bucle para crear  $n$  círculos, pero si queremos guardarlos tendríamos que añadir tantas variables como círculos, con el objetivo de retener cada conjunto de constantes. Este sistema es una duplicación del sistema para solo un caso.

La disgregación de los conceptos en datos y funciones tiene sus pros y sus contras, por ejemplo, este enfoque permite trabajar directamente con la idea de base de datos o archivo, lo cual puede ser beneficioso. Sin embargo, esta disociación implica **disminuir nuestro nivel de abstracción**.

#### 4.1.2. Enfoque orientado a objetos

El enfoque orientado a objetos es la forma particular de pensar el software en términos de elementos que colaboran entre sí para realizar tareas. Este enfoque nos da un nivel de abstracción superior al estructurado, asociando cada elemento del



problema a un elemento software. Cada elemento software tiene las propiedades íntegras de cada elemento del discurso (lo que *hace a una cosa ser una cosa*).

### Ejemplo: Pintar un círculo

En el ejemplo anterior, definimos un objeto `círculo` que cumple las propiedades de un círculo según la definición que hemos adaptado para nuestro sistema (en este caso, el objeto contiene un centro y un radio) y tiene los mecanismos para pintarse y crearse como elemento.

El enfoque de objetos piensa:

- En variables software capaces de recordar las constantes de un círculo, capaces de pintar un círculo y capaces de crearse a sí mismas como variables.
- En el sistema software en términos de la interacción de estas variables, dadas sus respectivas capacidades para ejecutar operaciones, es decir, cómo relacionar todas las variables para conseguir que se realice la tarea de pintar círculos.

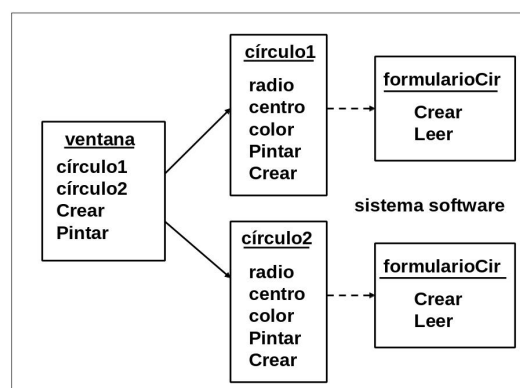


Figura 4.2: Sistema software con enfoque de objetos.

Este esquema muestra el sistema software, donde se aprecian las relaciones entre las variables software que ejecutan la tarea de pintar un círculo. Como vemos, el sistema software, aun aplicando el mismo algoritmo, tiene una organización diferente, y por tanto sus propiedades también varían.

#### 4.1.3. Objetos

Esto que hemos ido llamando *variables software* se conocen en este enfoque como **objetos**, y amplían la idea de la variable software tratada en el enfoque

estructurado, ya que tienen capacidad de expresar cualquier cosa, incluso operaciones. Otra definición complementaria de objeto es la siguiente: *Un objeto es un elemento software cualitativamente distinto capaz de expresar un concepto más amplio, más ambiguo: el concepto de cosa.*

Los objetos interactúan entre ellos mediante **mensajes**, solicitudes a objetos para que ejecuten operaciones. Una buena analogía de objetos y mensajes es el teatro, los objetos son *actores* y los mensajes son *su diálogo*; el programa describe a los actores, lo que tienen que hacer y decir.

## Propiedades de los objetos

Los objetos son definidos por su nombre, asociado con una dirección de la memoria de la máquina. Esta definición puede considerar además las propiedades que, generalmente, se clasifican en *atributos y operaciones*, también denominadas *métodos*.

- **Atributo:** Propiedad de un objeto, que está compuesto por un objeto o por una variable software tradicional. Los lenguajes OOP puros (como Small-Talk) solo admiten objetos como atributos, pero los lenguajes híbridos (como Java) también admiten variables software tradicionales.
- **Operación:** Propiedad de un objeto que expresa su capacidad para ejecutar la rutina indicada por la misma. Las operaciones representan cualquier código capaz de ejecutar acciones.

Los lenguajes de programación acostumbran a distinguir los atributos de las operaciones para elevar la eficiencia de compilación, pero en principio, **no hay razón para distinguirlos**.

**Globalidad de los atributos y operaciones:** Los atributos de un objeto son globales para todas las operaciones del mismo. Esto es, cualquier línea de código de cualquier método de un objeto tiene acceso inmediato a todos los atributos del propio objeto. Esto facilita el acceso a los atributos, pero tiene los siguientes inconvenientes:

- Cuando una operación de un objeto quiere usar otra operación del mismo, esta operación no conoce cómo usarla, ya que en la cabecera de la operación llamada no se refleja la relación con los atributos del objeto, y podría interceder en el uso de los atributos de la operación llamante.<sup>1</sup>

---

<sup>1</sup>Mutabilidad en los atributos del objeto.

- Al modificar alguna línea de código de una operación debemos revisar todas las líneas de código de todas las operaciones, ya que cambiar el comportamiento de una operación que a priori puede ser necesaria para el resto de operaciones podría desencadenar en un mal funcionamiento del objeto.

**Visibilidad de los atributos y operaciones:** Los objetos son, de una forma coloquial, *parcelas bien definidas*; accesibles desde dentro pero no tan fácilmente desde fuera. El acceso desde fuera está regulado por un control de visibilidad". Los elementos designados como *públicos* son accesibles a todos los objetos del sistema software. Los *privados* son accesibles sólo desde el propio objeto.

## Sobre la ambigüedad

Una de las características más importantes del enfoque orientado a objetos es la capacidad de éstos para expresar alternativas o significados diversos, lo que comúnmente llamamos **ambigüedad**.

El enfoque estructurado utiliza dato y función de transformación de datos como elementos del sistema software, mientras el enfoque de objetos utiliza los elementos objeto, con el significado de *cosa*, y mensaje con el significado de solicitud de servicio a una variable. El significado ambiguo de *cosa* que tienen los objetos nos permite hacer un diseño estructurado con aspecto (¿ropaje?) de objetos, pero no al contrario (diseño de objetos con aspecto estructurado).

**El enfoque de objetos se acomoda mejor a la diversidad de problemas que aborda el software.** Los objetos son más tolerantes para expresar la idea general de función que el enfoque estructurado, el cual está obligado a expresar esa función en términos de "entrada, proceso y salida". No obstante, esta mayor libertad de expresión tampoco es gratis siempre. A menudo hay que aceptar cualidades "extrañas" como la capacidad de pintarse, ampliarse, moverse, borrarse, etc. que tienen los círculos software, para ajustarse al enfoque.

La **ambigüedad del enfoque de objetos** también facilita que un **elemento software**, por sí solo, **exprese completamente un concepto**, por ejemplo círculo, mientras que el enfoque estructurado obliga, muchas veces, a disociarlos en funciones y datos.

La figura muestra la evolución del software hacia el aumento de la ambigüedad de sus elementos primarios. También nos muestra claramente qué es un TAD, es decir, un *tipo abstracto de datos*. Estos TAD **no se consideran objetos**, ya que,

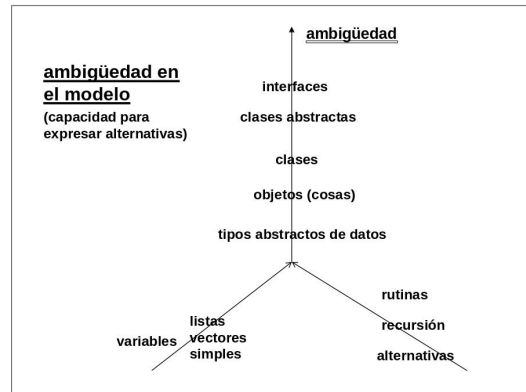


Figura 4.3: Aumento de la capacidad de ambigüedad en elementos software.

a diferencia de éstos, siempre deben ser datos (un objeto no tiene por qué ser un dato, puede ser cualquier cosa, no está comprometido a desempeñar dicha función ni ninguna en particular).

## Objetos vs Realidad

Gracias a la mayor capacidad expresiva de los objetos se dice a menudo que los objetos reflejan mejor la realidad. Pero esta idea perjudica, más que beneficia porque reflejar la realidad:

- No es el propósito de los sistemas software, salvo que sean de simulación.
- Dificulta los cambios en los sistemas software.
- ¿Qué significa? La realidad varía según el espectador.

## Refactorización

Se llama *refactorización* a los procesos que **modifican el sistema para mejorar alguna cualidad interna sin alterar sus funciones**. A diferencia de la forma tradicional, que aspira a alcanzar un diseño perfecto, la refactorización acepta que el objetivo es funcionar a tiempo, y que después de probarlo se podrá mejorar el funcionamiento del sistema software.

## Encapsulado y principio de ocultación

Como un objeto es una *parcela de software*, se suele decir que un objeto **encapsula** sus atributos y operaciones, lo cual es correcto, no distorsiona nuestra

analogía con la parcela. No obstante, cuando se asocia encapsulado con ocultación de información tenemos un problema.

Es cierto que el enfoque de objetos facilita algunas formas de desarrollo de software, como los prototipos evolutivos y el diseño en paralelo, pero no son facilidades intrínsecas.

**El principio de ocultación<sup>2</sup> establece que:**

- Cada módulo es independiente del código de la implementación de los demás (los clientes se comunican con los servicios por medio de interfaces, que permiten realizar las operaciones sin conocimiento de la modificación del código anteriormente mencionado).
- La interfaz o definición de cada módulo debe revelar lo menos posible del trabajo interno del módulo.

#### 4.1.4. Representación gráfica de los objetos

En este apartado veremos varios tipos de gráficos para representar objetos, sus atributos, operaciones y la interacción entre ellos (es decir, los mensajes).

#### UML (Unified Modelling Language)

UML es un sistema de símbolos y diagramas (un lenguaje) que sirve para expresar (modelar) los diseños de los sistemas software. La palabra unificado refleja el acuerdo de sus autores.

**Objetos:** UML representa a los objetos mediante **cajas**. Si se usa la definición extendida de objeto, la caja marca tres zonas: la superior, que contiene subrayado el nombre del objeto; la zona central, que contiene los atributos; y la zona inferior, que contiene los métodos u operaciones. En caso de usar la definición compacta de objeto, la caja solo contiene la zona superior. En UML los elementos públicos son diferenciados de los privados según el símbolo que precede su nombre; + para los públicos y - para los privados.

**Mensajes:** UML representa los mensajes mediante líneas dirigidas etiquetadas con el nombre de su operación, **desde el llamante al llamado**.

---

<sup>2</sup>Preguntado en el examen de 2017

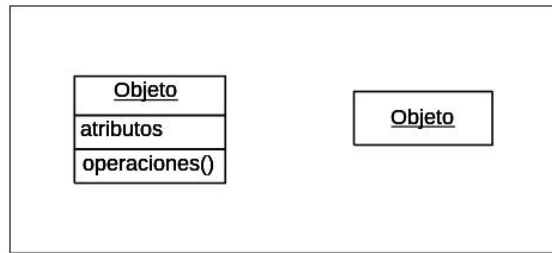


Figura 4.4: Representación de un objeto en UML.

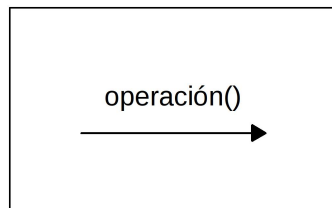


Figura 4.5: Representación de un mensaje en UML.

**Diagrama de secuencias:** Los objetos que participan en la interacción se dibujan, horizontalmente, en la parte superior del diagrama a través de sus esquemas simplificados (cajas conteniendo sólo el nombre subrayado). Debajo de cada objeto se dibuja una línea vertical discontinua llamada línea de vida que indica, en el eje tiempo, la existencia del objeto. Los mensajes se colocan entre las líneas de vida de los objetos, siguiendo la secuencia de ejecución. Si se quiere resaltar la devolución de algún valor, se puede dibujar una flecha discontinua apuntando hacia el objeto emisor, etiquetada con el nombre del valor de retorno.

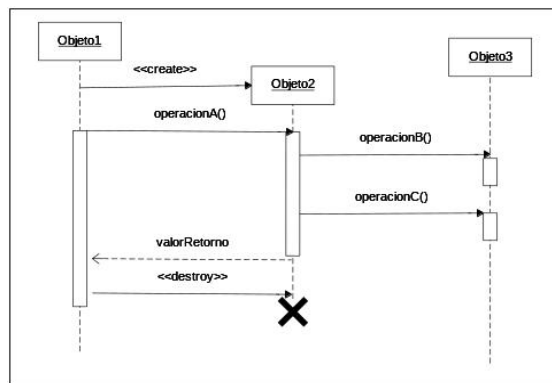


Figura 4.6: Diagrama de secuencias en UML.

Para indicar la creación o destrucción de un objeto se utilizan, respectivamente,

las etiquetas estereotipadas «create» y «destroy» en el mensaje que se le envía al objeto. La creación de un objeto se distingue dirigiendo el mensaje a la caja que representa al objeto. La destrucción de un objeto se muestra dibujando una X grande al final de su línea vida.

**Diagrama de colaboración:** Un diagrama de colaboración es un diagrama de interacción que resalta la organización de los objetos que envían y reciben los mensajes. Este diagrama muestra un conjunto de objetos, los enlaces entre ellos y los mensajes que intercambian.

**Un enlace es una instancia de una asociación o dependencia entre clases.** Se representa con una línea continua que une los dos objetos.

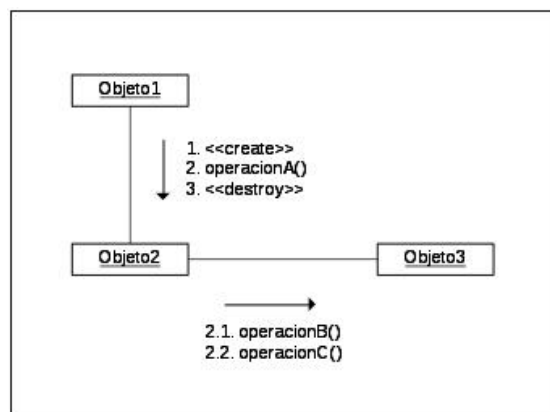


Figura 4.7: Diagrama de colaboración en UML.

Los mensajes se escriben junto a los enlaces, indicando el sentido con una flecha que apunta hacia el receptor y numerándolos para expresar el orden de ejecución. El diagrama de colaboración ofrece una vista de conjunto de estructura y funcionamiento, pero incide en la estructura, afectando a la claridad del funcionamiento (toca seguir la secuencia de mensajes saltando de una línea a otra, buscando la numeración).

Además, esta forma de representación dificulta la modificación del diseño, por lo que interesa usar este diagrama cuando no interese demasiado seguir el funcionamiento y se esperen pocas modificaciones del diseño.

#### 4.1.5. Clases

Una **clase** es una definición intensiva de un conjunto de objetos. Establece las propiedades distintivas de cualquier elemento del conjunto que define.

Considerando las clases se podría decir que un objeto es una **instancia** de una clase o un elemento del conjunto definido por una clase. La notación UML de un objeto en particular, teniendo en cuenta la clase a la que pertenece, es **objeto:clase**. Si se quiere expresar un objeto cualquiera (anónimo) de una clase, la notación sería **:clase**.

Las clases enriquecen el enfoque de objetos. Un objeto es una cosa y una clase define un conjunto de cosas con iguales propiedades, pero valores distintos, por tanto las clases establecen una generalización, una **abstracción** mayor que los objetos. La **estructura y relaciones de los elementos del sistema** software se pueden pensar en términos de las **clases**, y dejar los objetos para pensar los diseños dinámicos particulares del sistema.

La representación de las clases y los objetos coinciden prácticamente en el lenguaje UML, con la salvedad de que el nombre de las clases no se subraya.

## Diagrama de clases

El diagrama de clases expresa la estructura u organización del sistema software en términos de las clases. Además de intervenir en el funcionamiento, el diseño del diagrama es clave porque expresa la organización del sistema, y ésta decide sobre aspectos fundamentales: significado, facilidad de desarrollo en paralelo y facilidad de modificación.

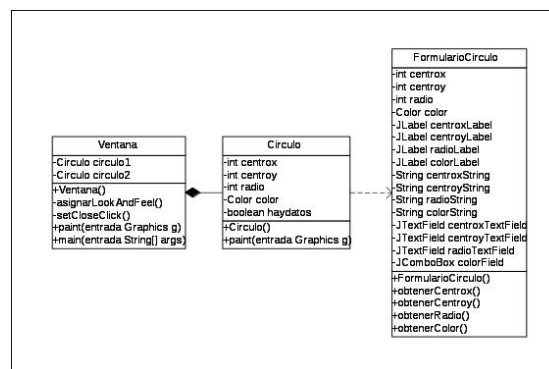


Figura 4.8: Diagrama de clases en UML.

La figura muestra el diagrama de clases del sistema para dibujar círculos. Se aprecian las clases, los elementos que componen cada clase y las relaciones entre las clases, así como la visibilidad de dichos elementos.

El diagrama de clases complementa a los diagramas de secuencias, muestra la



forma del soporte de los mecanismos, mientras que los diagramas de secuencias muestran el funcionamiento parcial de los mecanismos.

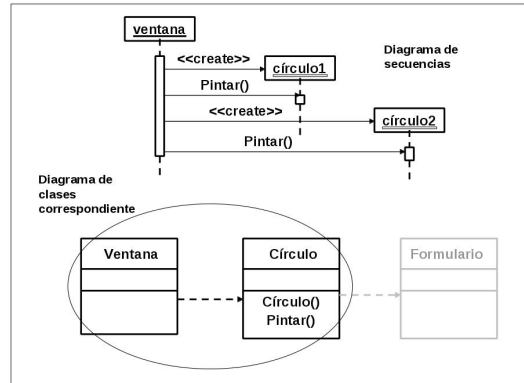


Figura 4.9: Correspondencia entre diagrama de clases y de secuencias en UML.

El diagrama de secuencias define el funcionamiento parcial del mecanismo para pintar círculos y el diagrama de clases correspondiente describe la organización de los elementos del sistema en términos de los conjuntos y sus relaciones.

### Relaciones entre clases

El enfoque de objetos acepta cuatro tipos de relaciones entre elementos<sup>3</sup>:

**Dependencia:** Se denomina *relación de dependencia* a la relación de una clase hacia otra cuando **los objetos de la primera usan objetos de la segunda**. Esta relación es direccional.

En UML la relación de dependencia se representa con una flecha discontinua del elemento dependiente A hacia el elemento independiente B (del que usa hacia el usado).

**Asociación:** Se denomina *relación de asociación* a la relación de una clase hacia otra cuando **los objetos de la primera contienen atributos que son objetos de la segunda clase**.

En UML la asociación unidireccional se representa con una flecha continua de la clase que contiene a la clase cuyos objetos son contenidos. La asociación bidireccional se representa con una línea continua entre las clases asociadas.

<sup>3</sup>Preguntado en el examen de 2017

**Agregación:** Se denomina *relación de agregación* entre clases cuando hay una **relación “todo/parte” entre los objetos de ambas clases**.

En UML se utiliza una línea continua y un rombo hueco en el lado del continente para representar una relación de agregación.

La relación de agregación introduce conceptualmente una estructura jerárquica del tipo ‘está formado por’ en el diagrama de clases, lo cual la diferencia de las relaciones anteriormente mencionadas (que son entre iguales).

**Composición:** La *composición* es otra forma de agregación con una fuerte relación de pertenencia: Un objeto “parte” sólo pertenece a un objeto “todo” y cuando se destruye el objeto “todo” se destruye el o los objetos “parte”.

En la composición el objeto “todo” controla la disposición y vida de las partes. Hay un cierto efecto transitivo: lo que le ocurre al todo le debe ocurrir a las partes. En UML se utiliza una línea continua y un rombo relleno en el lado del continente.

## Herencia

Desde el punto de vista formal, se establece que:

La *herencia* es una relación entre clases donde una de ellas comparte la estructura o comportamiento definido en una (herencia simple) o más clases (herencia múltiple). Las subclases pueden compartir atributos y operaciones de sus superclases, y también redefinir algunos de estos elementos.

Tenemos que tener mucho cuidado con la herencia, ya que, como ya hemos visto, generar dependencias fuertes es malo para el mantenimiento y la modificación de nuestro diseño software.

En un UML, la herencia se denota siempre desde la subclase a la superclase, con un triángulo hueco a la superclase.

## Polimorfismo

El polimorfismo es una de las cualidades más importantes del enfoque de objetos por su aporte a la ambigüedad en el diseño. Se asocia con el mecanismo de herencia y permite que la operación definida con la misma cabecera sea implementada de maneras distintas. Formalmente, se denomina polimorfismo a la **capacidad de una operación para manifestar un comportamiento diferente dependiendo del objeto que la ejecuta**.

## Principio de sustitución de Liskov

Después del análisis de los muchos problemas de la herencia, el principio de sustitución de Liskov, formulado hace casi dos décadas, ofrece un camino útil

y confiable para aprovechar los favores de la herencia. Literalmente el principio define, en términos de una sustitución segura, cuando una subclase es un subtipo de una superclase.

*“Si para cada objeto  $O1$  de tipo  $S$  hay un objeto  $O2$  de tipo  $T$  tal que para todos los programas  $P$  definidos en términos de  $T$ , el comportamiento [interno] de  $P$  no cambia cuando  $O1$  es sustituido por  $O2$ , entonces  $S$  es un subtipo de  $T$ .”*

Pero lo interesante es ver el principio de sustitución desde otra perspectiva. *La herencia permite sustituir un objeto por otro, es decir cambiar una tarea por otra, sin riesgo, siempre que las subclases sean subtipos de las superclases.*

El cumplimiento del principio de sustitución exige que los métodos de las clases derivadas deban mantener las siguientes relaciones con los métodos de la clase base:

- La clase derivada debe tener un método correspondiente a cada método de la clase base. Este método puede heredarse directamente de la clase base o sobrescribirse.
- Cada método de la clase derivada que se corresponda a un método de la clase base debe requerir lo mismo o menos que la clase base. Es decir, si se sobrescribe un método heredado de la clase base, las precondiciones del método deben ser más débiles o permisivas que las del método de la clase base. El nuevo método no puede ser más restrictivo que el método heredado.
- Cada método de la clase derivada que se corresponda a un método de la clase base debe garantizar lo mismo o más que la clase base. Es decir, si se sobrescribe un método heredado de la clase base, las postcondiciones del método de las clases derivada deben ser más fuertes o rigurosas que las heredadas de la clase base. Dicho de otro modo, el método de la clase derivada no debe comprometerse a ofrecer mayores resultados o resultados diferentes; sólo debe comprometerse a hacer lo que hace el método de la clase base, garantizando también las propiedades adicionales. Por ejemplo, si un método de la clase base devuelve un número mayor que el argumento que recibe, un método de una clase derivada podría devolver un número primo mayor que el argumento. Pero no estaría permitido que el método de la clase derivada devolviese un número menor o igual que el argumento.
- Está permitido que la clase derivada introduzca nuevos métodos adicionales que no aparezcan en la clase base.

## Clases abstractas

Se denomina *método abstracto* al método que sólo expresa la cabecera y carece de código interno. Dicho de otro modo, un método que está declarado pero no

implementado.

Se denomina *clase abstracta* a la clase que contiene al menos un método abstracto porque refleja la abstracción de ese método. Por ejemplo, la clase Figura. También se les llama clases virtuales o diferidas.

# Bloque 5

## Arquitectura

### 5.1. Introducción a la arquitectura

Requerimientos → Diseño → Construcción → Pruebas → Implantación.

La arquitectura de Software de un sistema es el **conjunto** de **estructuras** necesarias para **razonar** sobre el sistema<sup>1</sup>. Relaciona distintos elementos de software como son los objetos o los hilos de ejecución; el modelo del sistema, los diagramas, la lógica; o las entidades físicas como los nodos donde se ejecutará Software.

Desde una perspectiva de alto nivel, la arquitectura de Software cubre diferentes componentes del sistema. Tendrá en cuenta los requerimientos y fines del sistema, pero a la vez la creación del modelo, las dependencias y los escenarios de uso. Es decir, la arquitectura de software maneja sin encargarse de la implementación final, los aspectos técnicos y de uso que ocurrirán durante el desarrollo del sistema. Crea por tanto una estructura orientada al rendimiento, usabilidad y modificabilidad (**requisitos de calidad**).

### 5.2. Requerimientos

Objetivos de negocio → Drivers arquitectónicos → Decisiones arquitectura → Arquitectura documentada → Riesgos Deudas

Un requerimiento es una **especificación** que describe alguna funcionalidad, atributo o factor de calidad de un sistema software.

Requerimiento → Diseño → Documentación → Evaluación → Implementación

---

<sup>1</sup>Preguntado en el examen de 2017

Existe una amalgama de intereses y requerimientos entre los distintos actores que usarán el sistema. Si bien todos juegan un papel fundamental, a nivel de equipo de desarrollo se deberán satisfacer los requerimientos funcionales, es decir, usar una *combobox* para elegir los billetes. [1, p. 14]

La **ISO 9126** ofrece una descripción de los criterios de calidad del software (sección 1.5):

- Funcionalidad.
- Confiabilidad.
- Usabilidad.
- Eficiencia.
- Mantenibilidad.
- Portabilidad.

Los **drivers** son un subconjunto de requerimientos que definen la estructura de un sistema. Existen los drivers **funcionales**, **de atributos de alta calidad**, y los drivers de **restricciones**.

**Funcionales** Descomposición del sistema. Relevancia y complejidad.

**Calidad** Los atributos de calidad.

**Restricciones** Técnicas y de gestión.

### Métodos para identificar drivers arquitectónicos

Existen diferentes métodos para identificar drivers arquitectónicos. Podemos basarnos en *talleres de atributos*, métodos de diseño o *FURPS*<sup>2</sup>(*Functionality, Usability, Reliability, Performance, Supportability*).

El propósito de los talleres de atributos es ayudar a elegir la arquitectura adecuada para un sistema de Software. El modelo *QAW* (Talleres de calidad del Atributo) se centra en los requisitos del cliente, y no hace necesaria la existencia previa de una arquitectura software.

## 5.3. Diseño de estructuras

El diseño es la especificación de un **objeto**, creado por algún **agente**, que busca alcanzar ciertos **objetivos**, en un **entorno** particular, usando un conjunto de **componentes** básicos, satisfaciendo una serie de **requerimientos** y sujetándose a determinadas **restricciones**.

---

<sup>2</sup>Preguntado en el examen de 2017. Método que tiene en cuenta la usabilidad

*Teniendo en cuenta lo que nos han pedido, juntar piezas que tenemos teniendo en cuenta nuestras restricciones para describir lo que queremos hacer.*

Arquitectura→Interfaces→Detalle de los módulos

Se diseña en base a los principios de **modularidad**, **alta cohesión** y **bajo acoplamiento** y de **mantener las cosas simples**.

Los patrones de diseño juegan un papel fundamental en la especificación de los drivers. Se abstraen problemas ya resueltos sin llegar a representar soluciones detalladas para luego adaptarlo a cada caso particular. Cuando los diseños son más concretos, se llegan a crear elementos software reutilizables que proporcionan la funcionalidad genérica enfocándose a la resolución de un problema específico. Así nacen los **frameworks**.

A la hora de diseñar las **interfaces** se identifican los mensajes que se intercambian.

## 5.4. Diseño de Arquitecturas

El problema del diseño de la arquitectura se resuelve mediante diseños **basados en atributos**, **centrados en arquitectura** o con **vistas y perspectivas**. El método de **Rozansky & Woods**.

## 5.5. Documentación

*Generación de documentos que describen las estructuras de la arquitectura con el propósito de comunicar efectivamente a los interesados en el sistema.*

La documentación se apoya en vistas para la descripción de las estructuras. Se componen de un diagrama que representa los objetos de la estructura y de información textual que ayuda a comprender el diagrama.

La **vista lógica** representa en el diagrama *unidades* de implementación, que pueden ser en base a la funcionalidad o la responsabilidad.

Otras *vistas* son las de **comportamiento**, las **físicas** o la de Windows<sup>TM</sup> <sup>3</sup>.

## 5.6. Evaluación

*La evaluación es la técnica para evitar que los defectos lleguen a los usuarios finales o que se presenten en momentos donde corregirlos sea complicado.*

---

<sup>3</sup>Que no nos gusta.

	<b>ADD</b>	<b>ACDM</b>	<b>Rozansky &amp; Woods</b>
Mecánica y enfoque	Diseño iterativo descomponiendo elementos recursivamente.	Iteraciones de diseño, documentación y evaluación.	Iteraciones de diseño, documentación y evaluación.
Participantes	Arquitecto	Arquitecto y otros	Arquitecto y otros
Entradas	Drivers	Drivers y alcance	Vistas
Salidas	Esbozos de vistas	Vistas	Vistas
Criterios de terminación	Se satisfacen los drivers	Los experimentos no revelan riesgos o son aceptables	Los interesados están de acuerdo en que el diseño satisface sus preocupaciones.
Conceptos de diseño utilizados	Técnicas y patrones	Estilos arquitectónicos, patrones y prácticas	Estilos arquitectónicos y patrones.

Figura 5.1: Comparación de métodos de diseño de arquitecturas.  
Interesante ver la figura 1.1

La evaluación sirve para determinar si el software cumple con los criterios de calidad (1.3). Al evaluar un sistema se pueden producir *desviaciones* respecto a las necesidades de los usuarios o respecto a la construcción correcta del producto. Al evaluar las arquitecturas se busca satisfacer los drivers arquitectónicos (5.2).

## 5.7. Implementación

La implementación busca generar diseños detallados de los módulos y otros elementos siempre de acuerdo con la arquitectura. Se ajustan los diseños y errores, pero no se cambia la arquitectura.

- Diseñar la estructura del sistema basándose en la arquitectura.
- Basarse en los requisitos funcionales (5.2).
- Desarrollar<sup>4</sup>.

La resolución de las desviaciones (*errores*) se resuelve mediante controles de

---

<sup>4</sup>Picar código y fixes.



calidad en los que se **verifica el código**, el **diseño**. Además de realizar **pruebas** y **auditorías**.

## 5.8. Arquitectura Software

**Componente** Bloque del sistema. Parte que combinas con la arquitectura.

**Servicio** Funcionalidad que los componentes proporcionan a los actores.

Al dividir un sistema en componente, hay que definir los servicios que proporciona cada componente.

### 5.8.1. Estilos

El estilo es la forma general de un sistema, semejante a lo que serían los patrones de diseño (5.3). Al definir un estilo, se deben especificar los elementos como los bloques básicos de construcción, las conexiones entre los bloques y las reglas que especifican cómo se combinan los servicios.

Técnica	Patrón
Abstracción	Niveles
Encapsulación	Expedidor-receptor
Ocultación de información	Reflexión, Composite
Modularización	Niveles, Pipes & Filters, Composite
Acoplamiento y cohesión	Publicador-Suscriptor, Cliente-Despachador-Servidor
Separación de intereses	Modelo-Vista-Controlador

Figura 5.2: Patrones que ayudan a aplicar técnicas.

### 5.8.2. Índice de un documento de arquitectura

- Objetivos.
- Requerimientos (*funcionales, no funcionales*) (5.2).
- Decisiones y justificación.
- Modelo conceptual (5.5).
  - Modelo de componentes lógicos.
  - Modelo de procesos.
  - Modelo físico.
  - Modelo de despliegue.
- Despliegue de la arquitectura.

Otra información relevante del documento de arquitectura es presentar distintos diagramas:

- Diagrama de Clases (*Lógica*).
- Diagrama de Paquetes (*Desarrollo*).
- Diagrama de Interacción (*Procesos*).
- Diagrama de Despliegue (*Física*).

### **Pasos en la identificación de un problema**

Metas del proceso→Recogida de información→Conceptos de la Arquitectura→Cliente de la solución→Definición del problema

### **5.8.3. Patrones**

1. Especificar el problema.
  - Dividir el problema.
  - Encontrar el contexto.
  - Considerar pros/cons.
  - Acceder al catálogo de patrones.
2. Seleccionar la categoría de los patrones (*arquitectónicos o de diseño*).
3. Categoría del problema.
4. Comparar descripciones del problema.
5. Comparar beneficios y compromiso.
6. Elegir la mejor variante.

Entre los ejemplos de patrones están: *N-Niveles*, *Filtros y Tuberías*, *Pizarra*, *Modelo-Vista-Controlador*.

## Bloque 6

## Apéndices

### 6.1. Examen Junio 2009

Número de Matricula:

## INGENIERÍA DEL SOFTWARE II- FEBRERO 2009

- Duración total del examen: 2h:30m
- El examen consta de dos partes:
  - Parte A: Requisitos y Desarrollo Estructurado (3,5 puntos)
  - Parte B: Desarrollo Orientado a Objetos (6,5 puntos)
- El valor de cada pregunta se indica junto a su enunciado correspondiente
- Fechas aproximadas (si hay alguna modificación, se anunciará en el tablón de la asignatura y en la página web):
  - Publicación de las notas del examen: jueves 26 de febrero por la tarde
  - Revisión de examen: martes 3 de marzo a las 11:00

Preferentemente, las preguntas se contestarán en el espacio reservado para ello

### Parte A: Requisitos y Desarrollo Estructurado

- 1. (1 punto)** Explicar cada una de las cinco tareas de que consta el proceso de requisitos visto en clase (brevemente: una sola frase por tarea, y sin salirse del espacio reservado para la pregunta).

(De manera repetida, las 5 tareas son educación, análisis y negociación, especificación, validación y gestión de requisitos. Cada una de ellas se puede definir de múltiples formas, pero hay definiciones estándar en las transparencias de teoría de la asignatura.)

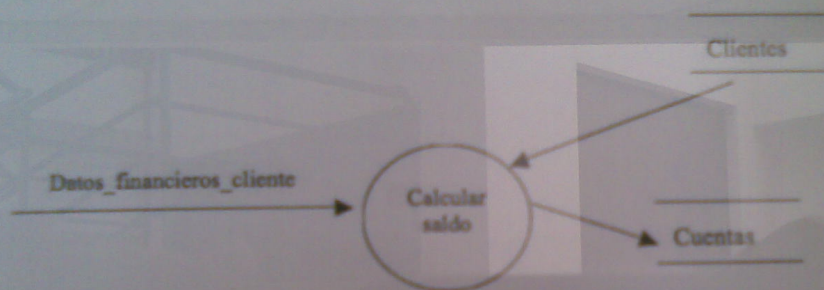
(De manera resumida, las 5 tareas son educación, análisis y registro de datos, gestión de requisitos. Cada una de ellas se puede definir de múltiples formas, pero hay definiciones estándar en las transparencias de teoría de la asignatura.)

**2. (1 punto)** Explicar si es posible (según la metodología de Yourdon) que dos burbujas se comuniquen directamente.

Sí, es posible, pero siempre por debajo del nivel de acontecimiento (es decir, puede haber conexión entre burbujas siempre que sean descendientes del mismo acontecimiento, como se ha visto en algún ejemplo hecho en clase).

Los autores de las publicaciones no tienen por qué ser usuarios del sistema. El sistema ya tiene previamente cargados los datos de todas las universidades y de las revistas.

4. (0,5 puntos) Este dibujo no es una representación válida de un acontecimiento. Explicar el porqué.



Faltan entidades externas (o terminadores), cuya presencia siempre es obligatoria en todo acontecimiento.



- Estancias en centros de investigación, indicando: nombre del centro, puesto ocupado, fecha de inicio de la estancia y fecha fin.
  - Comunicaciones presentadas a congresos, indicando: autores, título de la comunicación/ponencia, nombre del congreso, páginas de inicio y fin, lugar de celebración, fechas de celebración y lugar.
- b. Currículum vitae completo con el formato indicado en la convocatoria correspondiente. Dicho currículum será un fichero en formato pdf que se cargará en la aplicación.
- c. Hoja de servicios original del período del que se solicita evaluación (fichero en formato pdf que se cargará en la aplicación).

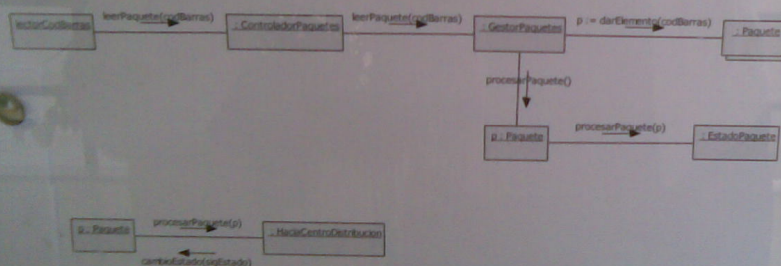
**(3,5 puntos)** En un sistema de logística los paquetes que se transportan pasan por una serie de estados diferentes. El ciclo de vida de un paquete se inicia cuando en el centro de atención telefónica se recibe una petición de recogida. El operador que atiende la petición registrará información acerca de la dirección de recogida (calle, número, piso, puerta, código postal, localidad), y el código de cliente. En ese momento el paquete se da de alta con el estado





En el sistema anterior, en diversos momentos a lo largo del ciclo de vida del paquete será necesario leer su identificación de paquete, a través de un lector de código de barras. Dependiendo del estado en que se encuentre el paquete, esta acción provocará distintas consecuencias. Así, por ejemplo, si se lee el código de barras de un paquete en estado "hacia centro de distribución", éste pasará al estado "en centro de distribución origen", mientras que si se lee su código de barras estando en este último estado, el paquete pasará al estado "en transporte por camión". Se ha decidido aplicar el patrón "Estado" para tratar esta situación.

- c) (0,75 puntos) Representar mediante un diagrama de interacción (de secuencia), el diseño de la operación del sistema *leerPaquete (codBarras)*, incluyendo la búsqueda del paquete correspondiente, y su tratamiento, según su estado. Representar explícitamente lo que ocurre cuando el paquete se encuentra en el estado "hacia centro de distribución", considerando que todos los posibles estados del paquete están ya creados, que cada estado sabe cuál es el siguiente estado, y lo único que hay que hacer es cambiar el estado actual.









### Parte A: Requisitos y Desarrollo Estructurado

- 1. (1 punto)** Explicar cada una de las cinco tareas de que consta el proceso de requisitos visto en clase (brevemente: una sola frase por tarea, y sin salirse del espacio reservado para la pregunta).

(De manera resumida, las 5 tareas son educación, análisis y negociación, especificación, validación y gestión de requisitos. Cada una de ellas se puede definir de múltiples formas, pero hay definiciones estándar en las transparencias de teoría de la asignatura.)

**(1 punto)** Explique si es posible (según la metodología de Yourdon) que dos bur

3. (1 punto) En un proyecto software nuestro equipo de desarrollo empleó demasiado tiempo diseñando, desarrollando y probando una función (llamémosle función F23) que, finalmente, no era del todo necesaria para nuestros usuarios/clientes. De hecho, empleamos tanto tiempo en implementar F23 que otras funciones (como F33, más importantes para nuestro usuario/cliente) quedaron, finalmente, relegadas. ¿Qué hemos hecho mal durante la fase de requisitos?

Hay múltiples formas de responder correctamente a esta pregunta, pero el factor que las une es el siguiente: no se ha hecho bien la priorización de los requisitos, lo cual ha llevado a un gasto excesivo de recursos en áreas que no eran demasiado importantes para el usuario/cliente.

### Parte B: Desarrollo Orientado a Objetos

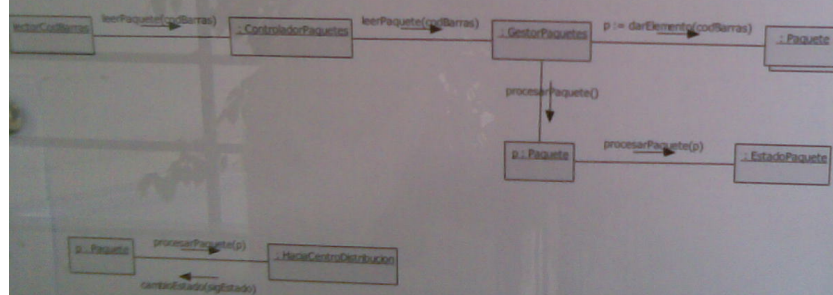
- 1. (3 puntos)** El Ministerio de Ciencia e Innovación desea desarrollar un sistema informático web que permita automatizar el proceso de solicitud de la evaluación investigadora para los profesores de las universidades públicas españolas. Dicha evaluación será de tipo voluntaria y con carácter sexenal. Para poder acceder al sistema será necesario registrarse. Para registrarse, el sistema solicitará al usuario su NIF, contraseña deseada y dirección de correo electrónico. A continuación, el sistema enviará un correo electrónico al usuario, confirmándole que ha sido dado de alta en el sistema. Una vez registrado, el usuario podrá hacer uso del sistema mediante su NIF y la contraseña que ha elegido. El sistema permitirá gestionar los datos personales de los usuarios (nombre, apellidos, categoría profesional y universidad para la que trabaja). Asimismo, el sistema permitirá introducir, modificar o borrar la información necesaria para completar la solicitud. Dicha información constará de:
- a. Descripción de la actividad investigadora durante los últimos seis años. Se habrá de incluir:
    - Proyectos de investigación financiados en los que se ha participado, indicando: nombre del proyecto, tipo de participación (investigador principal o miembro del equipo investigador), entidad financiadora, número de investigadores participantes, fecha de inicio y fecha fin.
    - Publicaciones realizadas, indicando: tipo de publicación (libro completo, capítulo de libro o artículo en revista), carácter (nacional o internacional), autores, título de la publicación, nombre del libro o revista (en caso de capítulos de libro o revistas), páginas de inicio y fin, editorial, año e ISBN ó ISSN (según sea libro ó revista).
    - Estancias en centros de investigación, indicando: nombre del centro, puesto ocupado, fecha de inicio de la estancia y fecha fin.
    - Comunicaciones presentadas a congresos, indicando: autores, título de la comunicación/ponencia, nombre del congreso, páginas de inicio y fin, lugar de celebración, fechas de celebración y lugar.
  - b. Currículum vitae completo con el formato indicado en la convocatoria correspondiente. Dicho currículum será un fichero en formato pdf que se cargará en la aplicación.
  - c. Hoja de servicios original del período del que se solicita evaluación (fichero en formato pdf que se cargará en la aplicación).



2. **(3,5 puntos)** En un sistema de logística los paquetes que se transportan pasan por una serie de estados diferentes. El ciclo de vida de un paquete se inicia cuando en el centro de atención telefónica se recibe una petición de recogida. El operador que atiende la petición registrará información acerca de la dirección de recogida (calle, número, piso, puerta, código postal, localidad), y el código de cliente. En ese momento el paquete se da de alta con el estado "pendiente de recogida", y se añade a la Lista de Recogida de Zona de la zona de distribución correspondiente. Para determinar a qué zona de distribución pertenece un determinado paquete se dispone de un componente que, a través de su fachada, ofrece un método *darZona(codigoPostal:cPostal): Integer* que, pasándole el código postal de una determinada dirección de recogida, devuelve el identificador de la zona.
- a) **(1,25 puntos)** Representar mediante uno o más diagramas de interacción (de colaboración), el diseño de la operación del sistema *altaPaquete* (*calle, numero, piso, puerta, codPostal, localidad, codCliente*), incluyendo la creación del paquete, su adición a la lista de recogida que le corresponda, y su asociación con el cliente correspondiente, teniendo en cuenta que más adelante será necesario poder acceder a todos los paquetes de un cierto cliente.

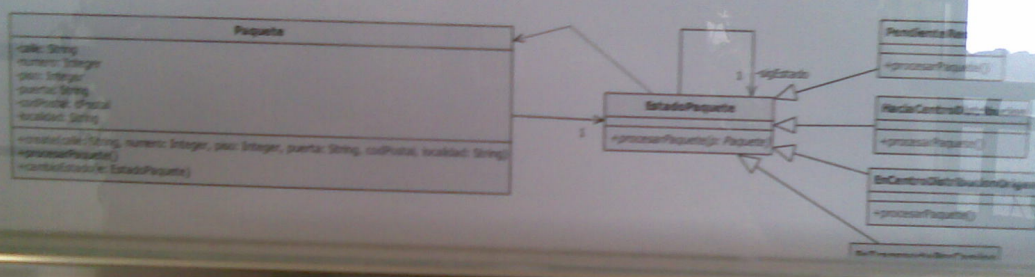
En el sistema anterior, en diversos momentos a lo largo del ciclo de vida del paquete será necesario leer su identificación de paquete, a través de un lector de código de barras. Dependiendo del estado en que se encuentre el paquete, esta acción provocará distintas consecuencias. Así, por ejemplo, si se lee el código de barras de un paquete en estado "hacia centro de distribución", éste pasará al estado "en centro de distribución origen", mientras que si se lee su código de barras estando en este último estado, el paquete pasará al estado "en transporte por camión". Se ha decidido aplicar el patrón "Estado" para tratar esta situación.

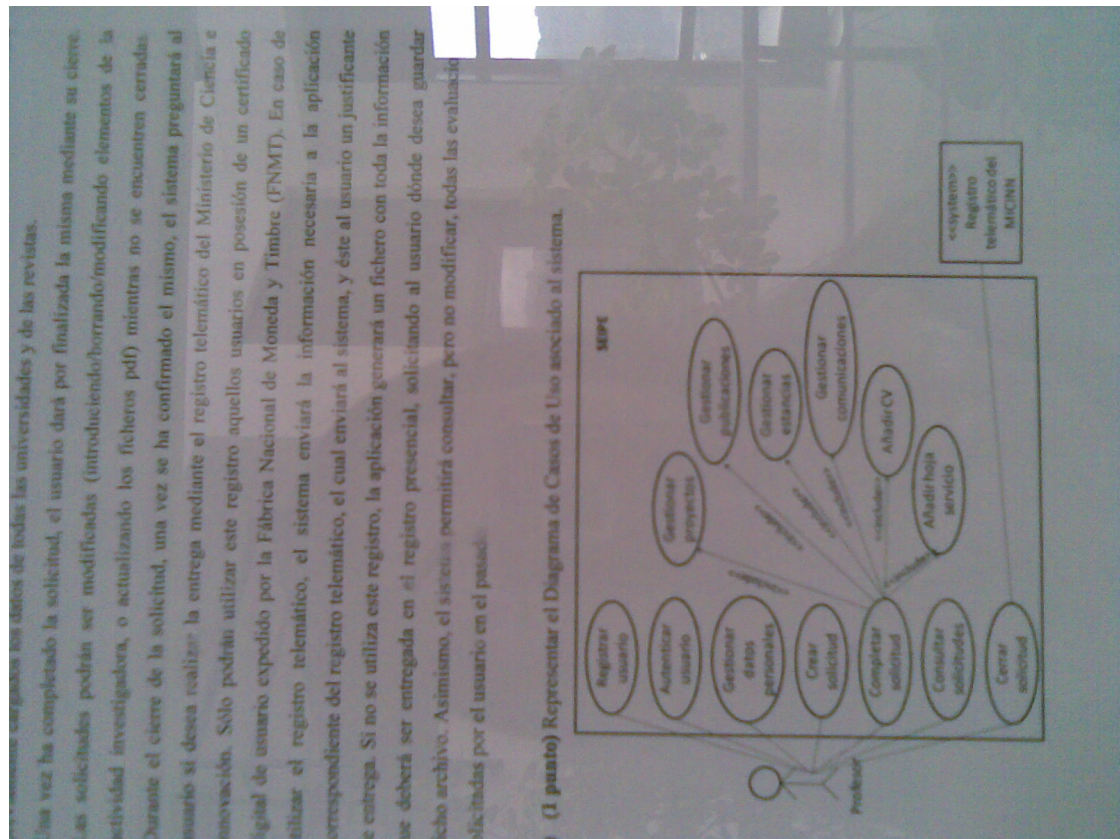
- c) (0,75 puntos) Representar mediante un diagrama de interacción (de secuencia), el diseño de la operación del sistema *leerPaquete (codBarras)*, incluyendo la búsqueda del paquete correspondiente, y su tratamiento, según su estado. Representar explícitamente lo que ocurre cuando el paquete se encuentra en el estado "hacia centro de distribución", considerando que todos los posibles estados del paquete están ya creados, que cada estado sabe cuál es el siguiente estado, y lo único que hay que hacer es cambiar el estado actual.

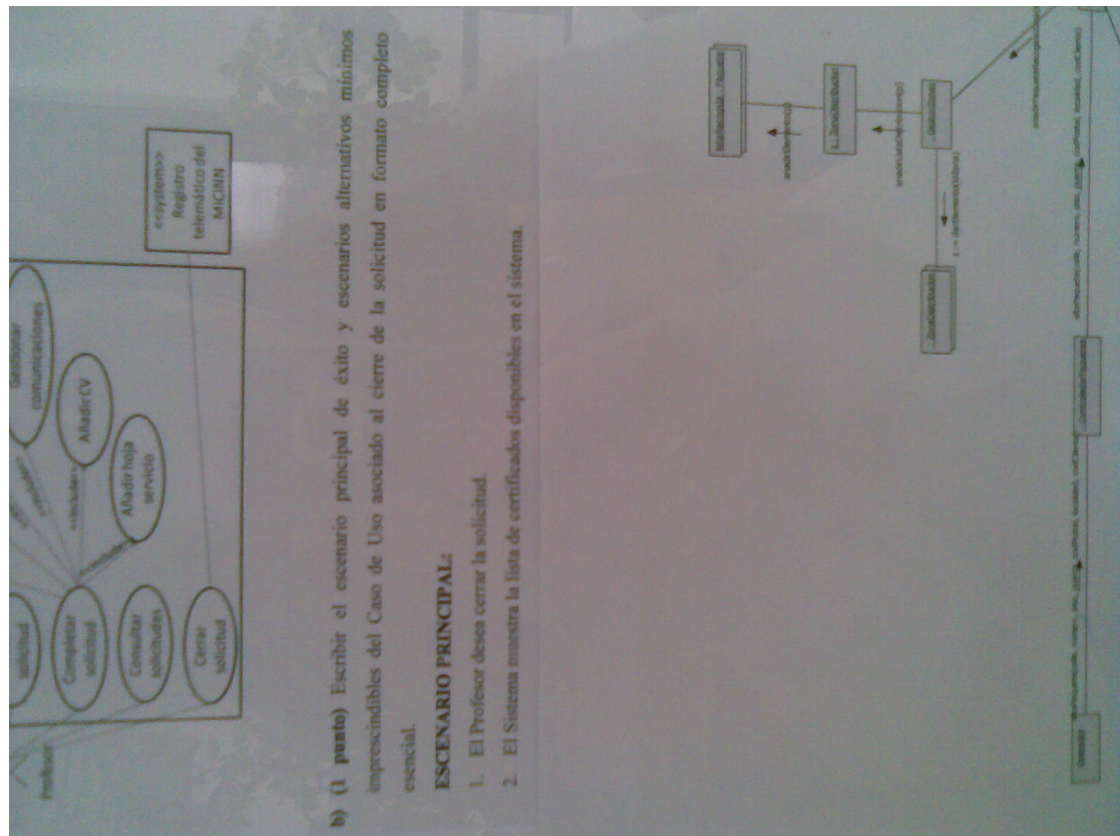




d) (0,5 puntos) Indicar las modificaciones necesarias en el diagrama de la pregunta b del ejercicio para representar este patrón.







# Bibliografía

- [1] Tomas San Feliu. Arquitectura software, conceptos y actividades, 2017. [https://moodle.upm.es/titulaciones/oficiales/pluginfile.php/325464/mod\\_folder/content/0/Material%20de%20Clases/2017\\_Conceptos.pdf?forcedownload=1](https://moodle.upm.es/titulaciones/oficiales/pluginfile.php/325464/mod_folder/content/0/Material%20de%20Clases/2017_Conceptos.pdf?forcedownload=1).

# Índice alfabético

ACDM, 38

ADD, 38

análisis, 15

baseline, 13

buen Software, 5

costes, 4

criterios de calidad, 37

drivers, 37

framework, 38

FURPS, 37

interfaces, 38

ISO 9126, 37

patrones de diseño, 38

QAW, 37

retos Software, 6

Revisiones de Validación, 12

Rozansky & Woods, 38

talleres de atributos, 37

trazabilidad, 12

vista lógica, 38